MechEng Software Engineering 3

Lecture 4 : Friday 29th January

Simon Gay
Department of Computing Science
University of Glasgow

2009/10

PSD3
Use Cases - Introduction

Ray Welland

6 October 2008

Project Elaboration

- The project has been funded, and we are now in a position to start work in earnest.
- The customer will provide a project description as the starting point, but the project development team will have to generate the important documents.
- They will usually have to obtain more information from the customer when drawing up the documents.
- We start with the following documents.
  - Project description.
  - Risk analysis.
- We will illustrate these features with the following example.

Part Time Teachers

- *We are developing software to management the employment, training and payment of part time teaching staff in a university or other such training establishment.*
- *Before the start of each semester the course directors produce a list of teaching requirements which we must try and fill. Our administrator will then attempt to find suitable staff and organise training for them.*
- *After teaching starts, we will receive a monthly claim form from each teacher, which we will verify and pass to the finance department for payment.*

Part Time Teachers (2)

- Note that we have been brief, talking about what we want to do rather than how we plan to do it.
- We have not worried about making the description perfect. The basic description is there, but we will certainly have to add more details later.
- This document is not produced by the customer and given to the system team, but written by the system team in conjunction with the customer.

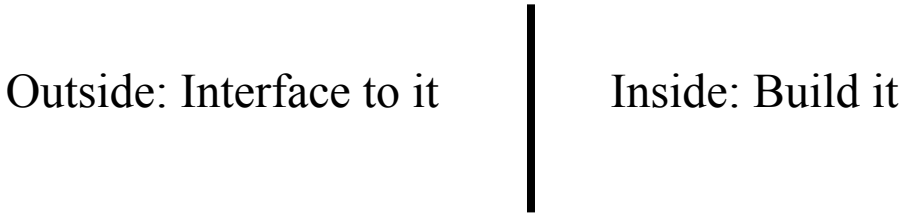
Risks

- Lack of user acceptance. The system must be easy for non technical people to use. (Requirements risk)
- Some of the people designing the software are inexperienced. (Skills risk)
- How do we handle the database crashing? (Technical Risk)
- We might not get the resources to complete the project (Political risk).

These are just the immediately important risks. We can add more later.

Identifying the System Boundary

- Now that we have a clear idea of what our system will do, we need to identify the system boundary and those things that lie inside and those that lie outside.
- If something lies inside the boundary then we have to create it.
- If something lies outside the boundary then it is already there, we do not have to worry about creating it.
  - we do have to interface to the external entities.
- We discover what the boundaries are by describing the actors and the use cases.

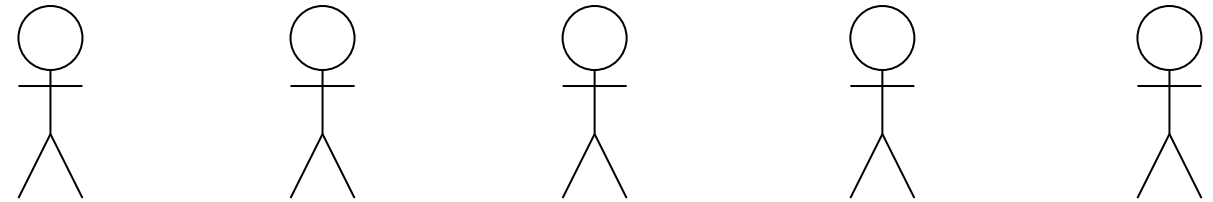


Actor Roles

- Objects outside the system are called actors. They initiate activities by using our system (active actors) or respond to activities (passive actors).
- Actors are anything that interfaces with our system. They can be people, other pieces of software, hardware, databases or networks.
- We are interested in actor roles rather than individual actors.
  - One person might take on several different roles, and hence be represented by several different actors.
  - Conversely, several different people may all perform the same role, and be represented by a single actor.
- Actors can also be other pieces of software or external systems.

Actors

- There will usually be several course directors, one for each course, but they will be represented by just one actor role, since all course directors interface with our system in the same way.
- One person may recruit people and also pay them. This person will be represented by two different roles.
- Actors are traditionally represented by stick figures with the name of their roles underneath.
- We might think of the following actors for our system.



Course Director | Teacher | Trainer | Recruiter | Payer

Actors (2)

- They are represented by stick figures. Each actor needs a short description defining his or her role in the system.

  ***Course Director**: A person who requires part time teachers for his or her course.*

  ***Recruiter**: Finds people to do the teaching.*

  ***Teacher**: A part time teacher.*

  ***Payer**: Organises payment for teachers.*

  ***Trainer**: Trains part time teaching staff.*

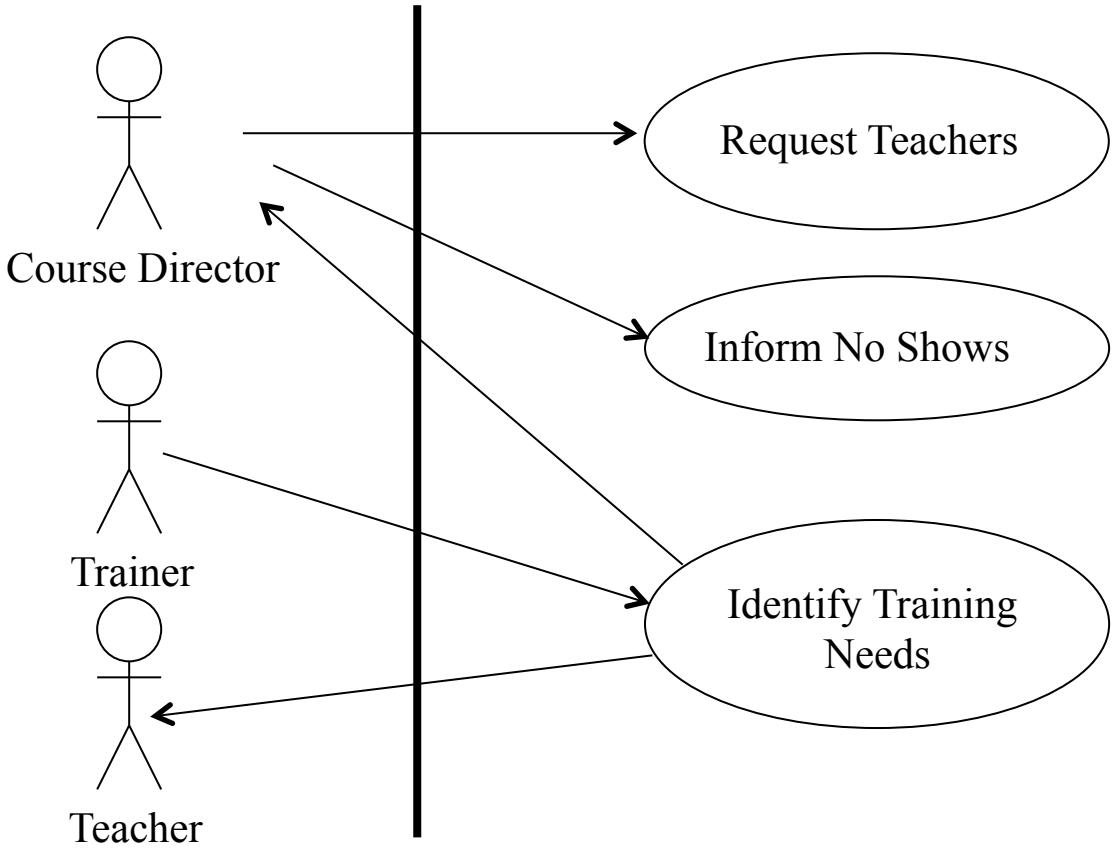
Actors (3)

- Notice how identifying the actors helps us to define the system boundary.
- Our system is not responsible for training the teaching staff, just for passing them on to the appropriate training organisation.
  - ***Trainer*** actor deals with training
- Similarly, we do not pay them ourselves.
  - ***Payer*** actor ensures payments are made

Use Cases

- Use cases define the internal activities of the system.
- We find the use cases by looking at the actors and seeing what activity they initiate.
- We record the use cases as ovals with a simple description.
  - this is the UML (Unified Modelling Language) notation, which we will use for this course.
  - we will extend the notation slightly in several places.
- Use cases should be short, doing one thing well, since it is less likely that they will be ambiguous.
- Let us look at three of the use cases associated with the Course Director and the Trainer.

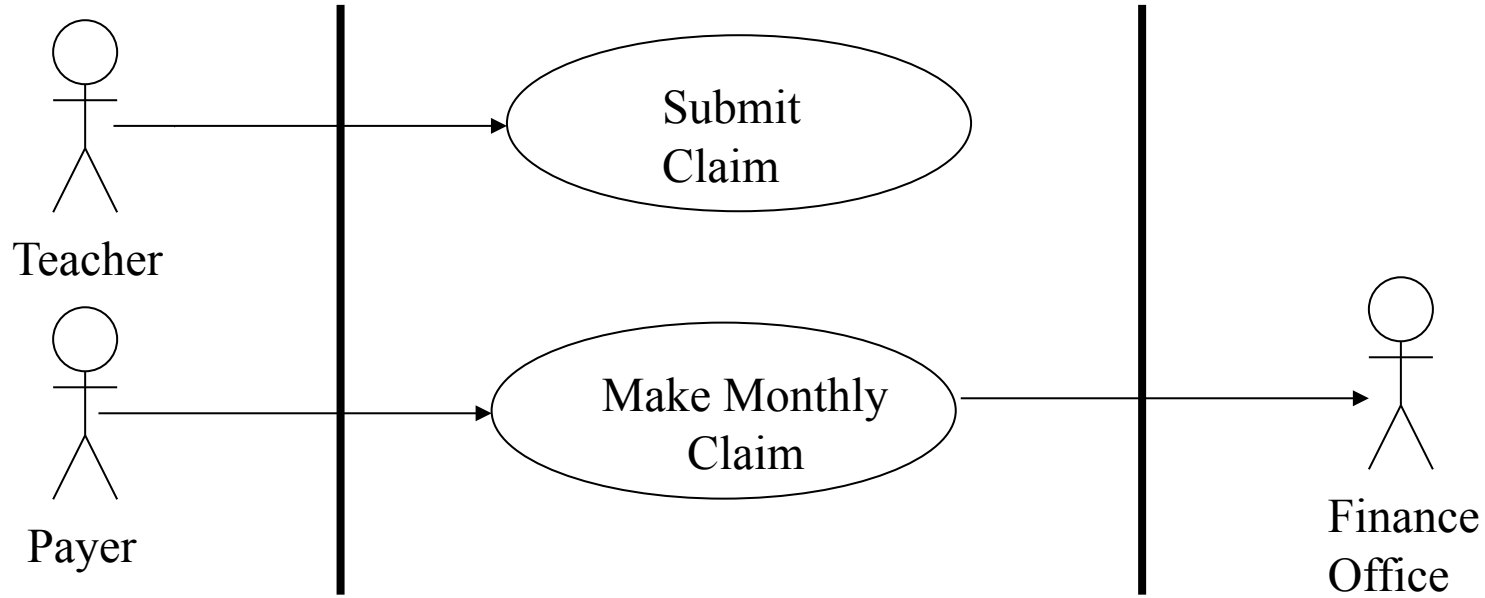
Use Cases (2)


Course Director
Trainer
Teacher
Request Teachers
Inform No Shows
Identify Training Needs

Use Cases (3)

- The arrow on the actor line indicates whether an actor is active or passive.
  - this will be different for each use case.
  - Course Director is active for some use cases and passive for others.
- Lines go from the active actor to the use case, and then from the use case to any passive actors.
- *This is a local extension to UML, which does not provide arrows on the lines connecting actors and use cases*.

External System as Actor



Assumes that Teacher submits claim to our system where it is checked and stored for further processing.

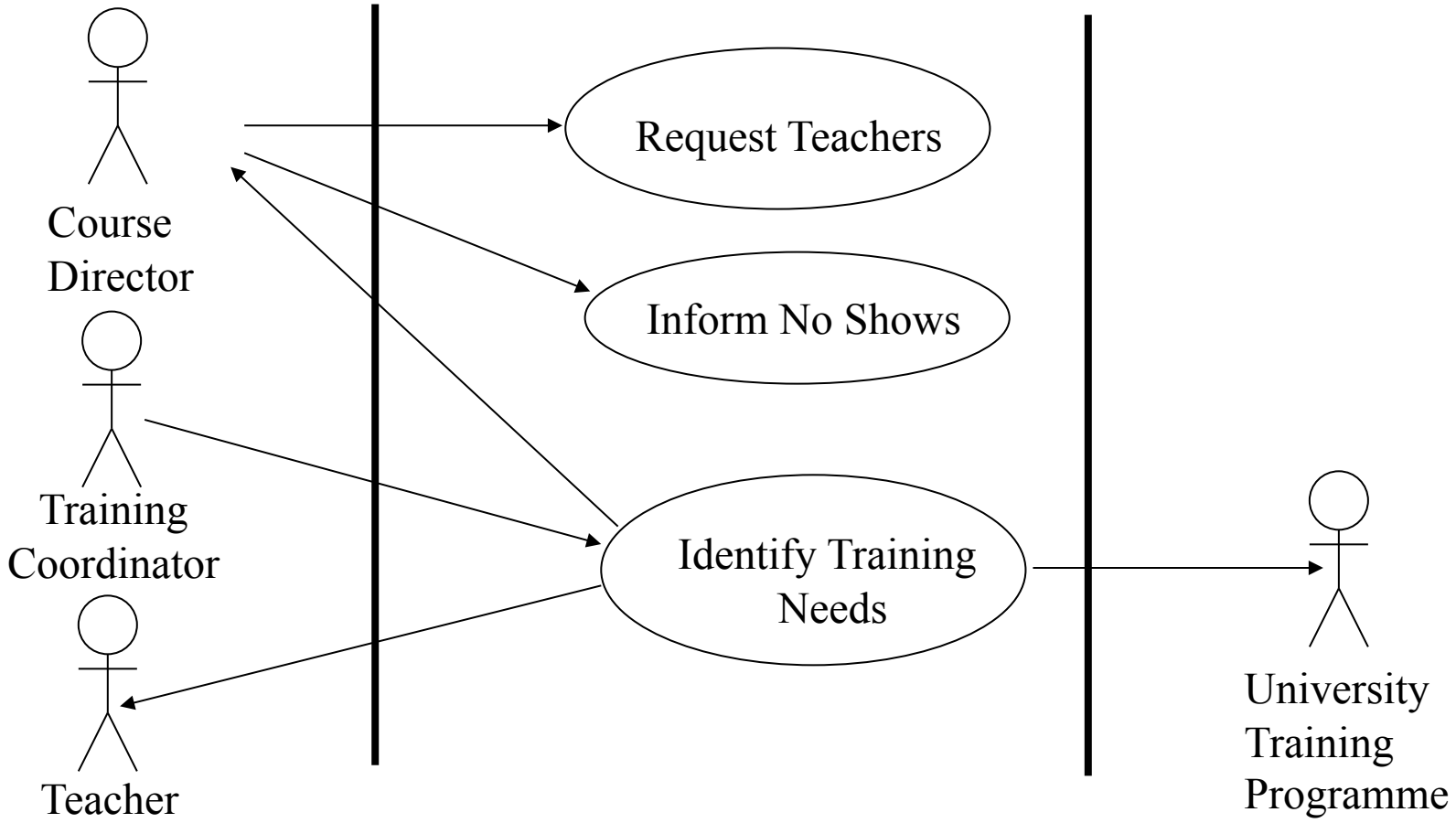
If claims are submitted by Teacher on paper to Payer then ‘Submit Claim’ is not a use case and should not be represented on the diagram.

Actual payments are outside our system.

Refining the Project Description

- These use cases have helped us to think about what the system will do and what it will not do
  - in this example, we could decide that we will not handle training of part time teachers, but delegate this work to the university.
  - this leads to an additional passive actor, the university system for training part time teachers.
- Another point of interest is that we do not record all of the activities of the actors
  - for example, the Course Director also organises courses, but we do not record this information because it is not part of our system.

Refining the Project Description (2)



Active and Passive Actors

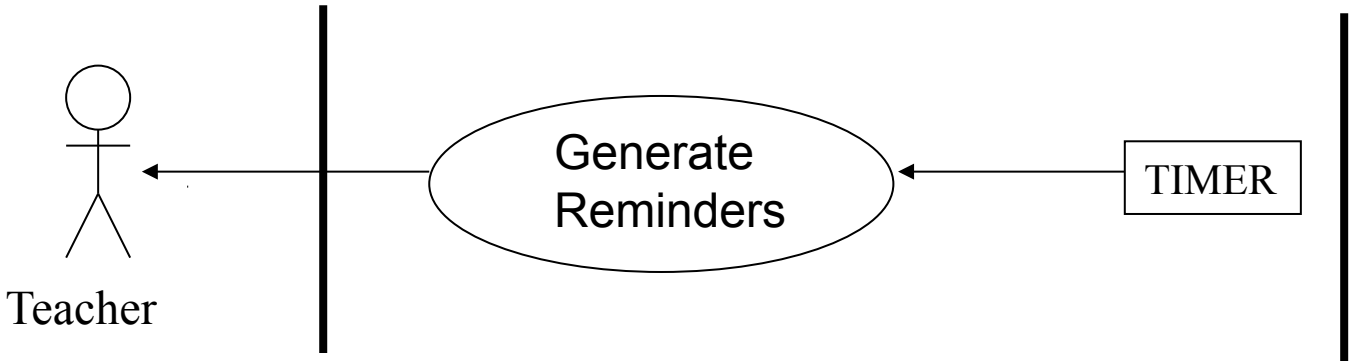
- Active actors will generate the use cases.
- If the actor is a person, think of him or her sitting down at a terminal and using a program.
- In our example, the Course Director is an active actor when requesting teachers for his or her classes.
- A passive actor is contacted by the use case, and does not initiate any activity.
- In our example, the Course Director is a passive actor in the "Identify training needs" use case. Think of them receiving an e-mail with a list of people, together with the training that is required.

Timed Activities

- Sometimes our system will generate activities internally.
- The use case will involve actors outside the system, but they are all passive actors.
- The most common way of doing this is by timed activities.
  - Some basic system functionality is scheduled to run later.
  - Teaching staff have to submit claims forms before the second Monday of a month in order to get paid at the end of the month.
  - We might send out automatic reminders.
- A common way of dealing with this is to have an internal TIMER actor, which activates use cases in the normal way.

TIMER

Timed Activities (2)



Boundary Problems

- Here are some common problems that arise when defining the system boundary.
- Are some of the requirements being handled by an actor?
  - Actors are outside our system and so cannot actually handle requirements. They will generate requirements.
  - We must redefine the actor role and generate new use cases to handle these requirements.

New Requirements

- What if we discover new requirements during the process of identifying actors and use cases?
- We should ask the following questions.
  - Are these requirements necessary for the system?
  - Are they something our system would naturally do?
  - Can they be handled by one of the existing actors?
  - How do they affect our current risk analysis?
- If we have added several new requirements we may have to step back and look at the complete system again.

The Scope of the Project

- The process of accumulating actors and use cases goes a long way towards defining the scope of the project.
- After a time we must step back and see what we have got.
- Are the system requirements all met by use cases?
- Examine the project description to see if we have covered everything.
- If there is something missing, then we need to go back and find some more use cases or actors.
- Alternatively, the use case approach may have generated more requirements, which can be added to the project description.

Non-Functional Requirements

- Some of the requirements cannot be met by defining use cases. These are non functional requirements.
- Typical examples are performance or security requirements.
- These non functional requirements can apply to the whole system or to each individual use case.
- For example, the Course Director must give at least one week's notice of teaching requirements, to allow for recruitment.
  - This is a non-functional requirement for the “Request Teachers” use case.
- The information stored must always be up to date and consistent at the end of each working day.
  - A system wide non-functional requirement.

Prioritising Use Cases

- Projects have start and finish date, with detailed budgets.
- It is often not possible to do everything that we want to, at least not immediately.
- We must prioritise the use cases to make sure that the important things get done and no effort is wasted on unimportant features.
- A typical priority scheme will have the following categories
  - Must have
  - Should have
  - Could have
  - Would like to have
- Must haves include not only the core functionality of the system but also items that it would be risky to exclude.